On A “Quantum” Proof Of The Equivalence of P And NP

Adewale Oluwasanmi

waleoluwasanmi@yahoo.com

Abstract

We present an algorithm, along with a correctness proof, for solving the 3 Satisfiability problem that is inspired by quantum mechanical principles and that runs in polynomial time with respect to the size of the input problem. Even though we term both our algorithm and its associated proof as quantum (for reasons which we will demonstrate), it is intended to be run on standard classical architecture. In the article, we posit that the 3 Satisfiability problem has an intrinsic complex quantum form that can be programmed in order to build a model of the solution space for satisfiable instances or show that such a model cannot be constructed. This yields surprising results on the ability for classical systems to abstractly simulate general quantum systems.

1. Introduction

The 3 Satisfiability problem is a popular problem in the field of computer science notable for being one of the quintessential problems in the NP complete space. The full classification of the NP space as well as the question of whether P = NP had the earliest developments in the combined works of Cook [2], Levin [3] and Karp [4]. In this article, we continue that analysis by providing a definite answer to this question by providing a “quantum” algorithm that solves the 3 Satisfiability problem in polynomial time. Our quantum dynamical approach is unique and has the closest analogue in existing literature to the holographic algorithms investigated by Valiant [1] and others.

In this work, we do not treat the problem in its regular logical form as a conjunctive normal formula over disjunctive unit clauses. We instead cast the problem into a novel quantum computational form or “type”. This treatment converts the initial propositional formula into a quantum programmable type (strengthening the propositions as types conception) whose particular type is algebro-combinatorial (an idea to be fleshed out in detail). This assumes that the reader is familiar, if needed, with the normal classically logical formulations of the problem in CNF form as well as how the maximal operational space of that formulation classifies the problem as NP complete in the literature.

The approach we present is intended to address the major source of uncoordinated exponential blow up or “drift towards infinity” in the classical approach, namely that attributable to the worst-case element of brute force search. To address this, we propose an alternative, “renormalizable” model where the original clauses are converted to a discrete solution form and the totality of potential and actual solution spaces are modelled as sets of “evolutions” of combinatorial species along with an

associated algebra of mutation/generator functions which extend the species by operating over their interference patterns.

The “algebraic” species are quantum “compressions” or representations of partial or complete solution spectra (if any solutions exist) and the generator functions construct complex “generations” of this space. In the case of unsatisfiable instances and instances with just one solution, the initial generation, termed first generation, is deemed complete. For instances with more than one solution, first generations serve as a complex differentiable base case over which other generations can be computed. All successive species constructed after the initial generation are collectively termed second generation species. Actual sets of satisfying assignments to unit propositions can be regarded as output/base species (that is they are of the same type, species, as the clausal space in our formulation and are complex homeomorphic with other species – if we regard all species as quantum deformable functions, with unit assignments as invertible constant functions as will be demonstrated).

The article is divided into 2 further sections as follows:

1. The Algorithm.
2. Conclusion.

The overall style pursued is functional in spirit as opposed to formal since our approach is algorithmic and relies less directly on formal methods established in more rigorous mathematical literature. Hence our main instruments of proof will be descriptions of structures and the functions that operate on these structures (algebraic) along with arguments to prove the correctness (soundness and completeness) of the entire procedure. A main requirement will be for the reader to be able to follow along and understand these (algorithmic/functional) structures, the operations that change them and the resulting (further) structures derived from applying the operations.

2. The Algorithm

2.1 Quantum Programs

We construe our algorithm as manipulating quantum programs, hence the term, quantum algorithm. These quantum programs, which we also term quantum injunctions, can be taken to be the programmatic equivalents of standard quantum computational elements, say qubits or qudits, that is, they are morphable and their morphisms can only take on standard quantum values. Ignore the fact that these programs themselves are the problems to be solved, that is, they extend classical quantum computation with the notion of self-solving quantum systems. This makes the programs a form of abstract quantum matter which, with some ingenuity, can be used to study other (physically concrete) quantum systems. Note that because we are dealing with a virtual form, these morphisms will always take on their desired quantum values, that is, we have no probabilistic element in the orientation or “spin” of the programs, providing an abstract and absolute form of topologically secure quantum computation.

2.2 Unit Variables and Literals

Unit literals represent traditional propositions that can be assigned true or false. Each literal presented can be identified by a subscript (natural number) and sign (+ for unnegated literals and – for negated literals). For brevity sake, we only show signs for negated (-) literals and assume that unsigned numbers are positively signed. Every literal and its opposite (inverse) are antipodes the same variable (simply identified by the lower letter "x" attached to their common subscript) which is their algebraic "superposition". The signed representations (literals) are the only elements directly available to functional manipulation but we will continue to use the term "variable" to refer to the abstract sense of assignment so we can talk of assigning a variable, say $x_{3,}$ to one of its assignable literals, 3 or -3 , that is, we make that particular literal true and the opposite literal (same variable, opposite sign), false. The selected notation for literals is similar in spirit to the DIMACS CNF format used by many classical SAT solvers.

2.3 Basic Quantum Program Structures

Each initial disjunctive clause is represented by a Quantum Injunctive Clause, which we will call an X structure. This is a comma separated list of literal values, where the integer part represents the variable subscript and the sign is the OPPOSITE of that carried by the literal in normal CNF format, nested within opening and closing braces. For example:

Clausal form $(\neg x_1 \vee x_4 \vee \neg x_7)$ is transformed to X structural form {1, -4, 7} and

Clausal form $(x_3 \vee \neg x_5 \vee x_6)$ is transformed to X structural form {-3, 5, -6}.

We will call all X structures that directly represent the clauses in the original formula, the BASIS X structures. We will shortly see how this basis is extended to compute the complete satisfiability model of the formula.

The entire CNF formula itself is represented by a different structure termed a Y structure which is a comma separated list of all (valid) X structures. We will shortly explain the notion of a valid X structure. This gives Y structures a richer, computationally effective (functional versus formulaic) structure. For example:

The formula $(\neg x_1 \vee x_4 \vee \neg x_7) \wedge (x_3 \vee \neg x_5 \vee x_6)$ is transformed to Y structural form {1, -4, 7}, {-3, 5, -6}.

The last class of structures, Z structures, are the actual satisfying solutions to some problem instance S. A Z structure is represented as a comma separated open list (no opening or closing braces) of assignments (literals that have been made true). For example:

The Y structure: {1, -4, 7}, {-3, 5, -6} is satisfied by the Z structure: -1, -4, 7, 3, 5, -6.

Note how in our example, only the first literal in each Y structure has its sign reversed in the Z structure and is the only assignment that satisfies the respective underlying disjunctive clause (because, as we have explained, X structures reverse the sign).

Signs are reversed in X structures for the following reason: Computationally, X structures can then be interpreted as declared conditional programs for which the following axiom must be satisfied:

> **Axiom 1**: "For some X structure, X1, composed of n distinct elements and for any program P1 with an attached, partially completed Z structure, Z1, that satisfies (must satisfy) the given instance S containing X1, if n-1 of the literals in X1 have already been added to Z1, that is, these n -1 variables in Z1 have already been assigned to literals which do not satisfy the classical disjunctive clause represented by X1, the program P1 must immediately assign the nth, unassigned literal, call it k, in X1 to its opposite Thus, we extend Z1 to a new Z structure, Z2, that satisfies X1. We call Z2 a satisfying completion of X1 over k".

For example:

> The Y structure: {1, -4, 7}, {-3, 5, -6} has the partially completed Z structure: -1, -4, 7, 5 at some time t and at the successive time t + 1 we extend the solution to -1, -4, 7, 5, -6 by adding -6.
>
> Note that the second X structure is unsatisfied at the new time t + 1 and that we must immediately at what we call time t + 2, add 3 (the opposite of -3) to the Z structure, obtaining the satisfying solution: -1, -4, 7, 3, 5, -6.
>
> At time t + 2, we can regard the previous n–1 unsatisfying literal assignments as the INPUT portion for the inclusion of the satisfying literal assignment (the OUTPUT) of the anonymous function locally defined (in space and time) on the X structure. This qualifies us to call each structure X of size n, an n-dimensional program encoding n functions of input size n-1 and output size 1.

Axiom 1 treats X structures as algebraic types from which we can construct a new type via pattern matching. Since Y and Z structures vary along with X structures, they can be regarded as algebraic types themselves. Speaking algebraically then, we think it worthwhile to make the following section closing statements by speaking of the "cycle" structure of the 3 algebraic types:

> Every completed satisfying program P1 outputs a permuted/chained variation of a permutable set of assignments C (that is, the elements of C should be assignable in any order). We say each X structure carries a partial cycle on C chains.
>
> Every X structure of size n that carries a cycle on some combination C, over some output literal j, also carries a (exactly n-1) cycles on other C structures for which the negation (contra-variation) of j is an input assignment. Thus, every X structure is a co-cycle over its n variants, that is, every X1 structure is a partial co-cycle over its n associated C structures.

On the other hand, Y structures must necessarily carry a total co-cycle over every possible combination (where solutions exist), at every instance.

Z structures are just particular chain realizations of some covered combination C.

2.4 Injunctive Space Completion

Here, we introduce an additional axiom and its corollary that both define a partial binary "integration" operation over X structures.

Axiom 2: "Take any 2 valid (again, we will explain invalidation rules shortly) X structures, X1 of size m and X2 of size n. If X1 and X2 share a common variable x_i, such that x_i takes on opposite signs (literals) in both structures AND X1 and X2 share no other variables, DIFFERING IN SIGN, we must generate and add to the global Y structure, a new X structure X3, composed of the m-1 elements in X1 and n-1 elements in X2, differing from x_i".

This can be read more procedurally as: "If Y is to contain only valid X structures, then it must admit only X structures that do not lead to assignment contradictions when applied in parallel since by Axiom 1, if we do not construct X3, x_i will be assigned two contradictory values if it is the last value assigned in both X1 and X2 by some arbitrary program P". Thus, Axiom 2 defines a binary parallelization/synchronization operator on our X structural programs.

Corollary 2.1: Self Contradiction/Reduction: "Take 2 structures X1 and X2 both of size n having the same set of variables and in which n-1 of the variables are the same and agree in sign (same literals) and also having an nth common variable that differs in sign (opposite literals). We must generate a new structure X3, composed of the agreeing n-1 literals. X1 and X2 are now invalid as they have been replaced/reduced by X3, a valid substructure of each.".

2.5 First-Generation Algorithm

i. Take a given 3 SAT problem instance S, convert all of its clauses into X structures and add them to a list called "incoming".

ii. Create a new list called "processed" to add structures that have been picked up from "incoming" and processed as described in step iii.

iii. Process elements of "incoming" in a loop, and for each iteration:
Remove the first "incoming" structure X1 and compare it to each applicable element in "processed" using Axiom 2.
If we generate a new X structure during a comparison with 5 OR FEWER elements, add it to "incoming", else discard it (NOTE THIS STEP).
Add X1 to "processed.
Continue until all elements in "incoming" is empty.

iv. If at any point, we generate two X structures X1 and X2 each of size 1 and each containing the same variable but where each variable is the literal opposite of the other, X1 and X2 are invalid and S is unsatisfiable, otherwise, if we exhaust "incoming" and no such occurrences are recorded, S is satisfiable.

2.6 Proof of Correctness

It should be quite obvious from the description above that we are dealing with a polynomial time algorithm (all considered X structures are bounded by a finite size of 5) with immediately evident linear reason why the procedure is correct. The best way to prove the correctness of this algorithm will be to examine its edge cases.

First, we must agree that if we complete our first generation such that by our criteria above, the problem instance S is satisfiable (to be proven), then all valid X structures describe necessary actions that must be taken at "boundaries" by any satisfying program P according to Axiom 1.

Secondly, during second generations, as P assigns variables to literals, any X structure can be rewritten/rescaled with a structure reflecting all completed assignments. For example:

Say we have some X structure {3, -5, -7}.

If we assign the variable x_5 to false, that is, our Z structure contains -5, we can perform the following rewrite:

{3, -5, -7} to {3, -7}.

This ensures that the X structure indicates the new exact boundary rule that satisfies the underlying disjunctive clause.

This is what we mean by X structures being quantum computational elements that can be reprogrammed by taking a quotient on a structure to obtain a valid (and necessary) quantum "adjoined" boundary rule.

Henceforth, we will continue to use particular X structures with clearly defined literals (1, -1, 2, and so on) to demonstrate all described operations in the proof. As the reader will see, this use is abstract enough to demonstrate all necessary points.

Next, consider any two X structures of size 2 such that X1 = {1, 3} and X2 = {1, -3}. Clearly, the variable x_1 cannot be assigned a positive value 1 at any point in the program because this leads to an unsatisfiable condition/assignment dilemma on x_3, as x_3 then cannot be assigned either a positive or negative literal value.

The main line of the proof will show that upon successful completion of a first generation on some arbitrary 3 SAT instance, we never run into the assignment dilemma mentioned (in a very exact, abstract sense which we will demonstrate). This fact couples with the additional fact that because our X structures always denote only valid actions that must be taken at "boundaries", only invalid solutions are ever excluded at each point in time. The result is that our proof is able to show that we can after first generation, continuously assign values to all variables, where some assignments may be free and others bound (to the value necessitated by the boundary rule indicated by some valid X structure).

The approach taken can be used to show that our algorithm is sound and complete (at every point in time) with respect to satisfiable instances. The details are a little subtle and so we ask the reader to pay close attention both individual steps and their combined effect in order to validate the reasoning.

Henceforth we will convert the term first generation into a single formal word – first-generation (note the hyphen). We will also apply this compression to the associated term second generation which will be referred to as second-generation.

Here are the edge cases that prove our approach:

Case 1:

X structures {1, 3} and {1, -3} exist as part of the first-generation completion.

By Axiom 2, this cannot be the case at all since we will have combined both structures into the 1 element "constant" structure {1}.

Case 2:

X structures: {1, -3}, {1, 2}, {-2, 3} exist as part of first-generation completion.

Here assigning the variable x1 to 1 forces the following other assignments by virtue of Axiom 2 (X structures to the left of colon: assignment implications to right of colon):

{1, -3} :3

{1, 2}: -2

{-2, 3}: -3

But by Axiom 2, this cannot be the case as the following actions would have been taken in the first-generation stage:

{1, 3} and {-2, 3} would have yielded {1, -2} which would combine with {1, 2} to yield just {1}.

In this case, this means that the only assignable literal to x_1 is -1 (the inverse of 1).

Case 3:

{1, 3} and {1, -3} exist as part of a rewrite of two independent X structures, X1 and X2 each of size 3.

This means we can't assign the literal 1 to true. We call this situation a "local" assignment blockage on assignment to the literal 1.

This can happen say we originally had X1 = {1, 3, 4}, X2 = {1, -3, 5} and then assigned both 4 and 5 to true.

By axiom 2 however, during first generation, we would have constructed an extra structure X3 = {1, 4, 5}, such that on adding 4 and 5 to the Z structure of a program, we will be forced to add -1 by virtue of Axiom 1.

The question we then need to ask is, does the selection of 4, 5 and -1 lead to a dilemma over another variable, for example, say we have the following 2 structures:

{-1, 4, 5, 6} and {-1, 4, 5, -6}.

Why this (and just this question) question? – In the process of Z structure completion, we posit that we are only interested in guaranteeing continuity "local" variable changes, that is in situations where we

explicitly assign a single literal (a variable antipode) to true or are forced to assign it to true. And the only times we are "immediately" forced to a assign any literal to true is at the level of literal assignments over 2-element X structures, say:

We have {1,2} and we select 1 (assign it to true), then we immediately have to select -2.

In the case of our example where we have {-1, 4, 5, 6} and {-1, 4, 5, -6}, the selection 4 and 5 would immediately create the structure {-1, 6} and {-1, -6} (which we will call conditionally implied 2-element structures) which would then also render us unable to assign -1 to true.

In our case however, the presence of our two offending structures: {-1, 4, 5, 6} and {-1, 4, 5, -6} would have caused the construction of a new X structure {-1, 4, 5} during first generation. This would have combined with X3, that is, {1, 4, 5} to create {4, 5} which means that 4 and 5 could not be simultaneous assignments in any valid program and we would not have a blockage on either 1 or -1 (no local blockages on the 2 antipodes of any variable).

Be convinced that 4 and 5 in this case need not necessarily be different literals, but by virtue of their difference, show the maximal extension of this case.

We now add a new axiom as a result of our analysis of this case:

Axiom 3: "After rewriting any two 3-element X structures, if we derive from these rewrites, a local blockage on assignment to a literal x, our program can assign the negation of x to true, that is, add the negation of x, which has no local blockage, to the current Z structure ".

Case 4:

Variation on case 2 for rewrites.

We have following X structures as a result of rewriting three different 3 element structures: {1, -3}, {1, 2}, {-2, 3}.

As shown in case 2, this also generates a dilemma for assigning x_1 to the literal 1. Let us again, use a maximally extended case in which the 3 different structures use 3 different extra variables:

X1 = {1, -3, 4}, X2 = {1, 2, 5}, X3 = {-2, 3, 6}

On first-generation, applying axiom 2, we would have generated the following X structures:

{1, -2, 4, 6}, {1, 3, 5, 6} and {1, 4, 5, 6}.

It is again clear that assigning 4, 5 and 6 will explicitly force us to assign x_1 to -1.

Again, as in case 3, the follow up question we need to ask (for local variable assignment coverage) is, if only {1, 4, 5, 6} exist and not {-1, 4, 5, 6}, combinatorial-wise, how does the selection of 4,5,6 and -1 "act" on the space. Specifically, is it possible to have the following two 5 element structures:

{-1, 4, 5, 6, 7} and {-1, 4, 5, 6, -7}?

The existence of these 2 structures would mean that we could not assign -1 as expected in addition to 4, 5 and 6 since we would then not be able to select any value for X_7 and would therefore need to generate a new structure {-1, 4, 5, 6} which would combine with {1, 4, 5, 6} to yield {4, 5, 6}.

Now, let us revisit the rewritten example for our case: We have following X structures as a result of rewriting three different 3 element structures: {1, -3}, {1, 2}, {-2, 3}.

Similar to case 3, we call this particular example a local assignment blockage on assignment to the literal 1.

Again, as in case 3, We add a new axiom as a result of our analysis of this case:

Axiom 4: "After rewriting any three 3-element X structures, if we derive from these rewrites, a blockage on assignment to a literal x, our program can assign the negation of x to true, that is, add the negation of x, which has no local blockage, to the current Z structure".

Case 5 (terminality case):

Note that we have proven up to this point that any assignment blockages we encounter upon a rewrite of up to three 3 element structures can be resolved by reassigning the blocked literal to its negation.

Please be convinced that 3 is the maximal number of 3-element structures we need to consider in a single step for analyzing local blockage on any literal that can result from rewriting any number of 3-element structures (that is, case 4 captures the maximal blockage pattern on any such literal).

Please be also convinced that we need to only consider actual and "implied" local blockages in our analysis, that is, we can always guarantee that no two opposing literals of the same variable are both locally blocked at any point in time and that there is a guaranteed local continuity of assignment over every variable point. This is a notion of continuity that has to be fully and intuitively grasped in order to ascertain the full correctness of our procedure.

We now add a closing axiom:

Axiom 5: "Any program which can continuously rewrite all the BASIS injunctions that directly represent the original clauses (which all contain at most 3 elements by natural transform) such that none of them assigns a literal to true which is assigned false by another, is sufficient to prove that the original formula is satisfiable".

If we agree that all our axioms are sound, then we can also conclude that since all structures only exclude invalid assignments at all points in time, then our algorithm is also complete, and indeed we need no more than structures of a maximum of 5 elements to fully determine the satisfiability of our original clause.

One last set of non-crucial points to observe are:

1. Freely assignable variables not bound to a represented structure may occur at any point during the course of running the algorithm. We will call these structures radicals and
2. 1 element structures (constants) may occur during first-generation prior to second-generation selections and that in fact all variables over the instance signature may collapse into just one

assignable literal each, generating only a single combination instance, in which case there are no possibilities for second-generation modifications. This is a closed loop.

3. If we accept that our algorithm is sound and complete, assignments need not perform a reduction on any structures but simply treat all structures as “concurrent” functional programs that always agree on their combined outputs.

Combining all said proof points, we conclude that if we can’t construct a first-generation model (dilemma over some arbitrary variable), our instance S is unsatisfiable, otherwise it is satisfiable and we can generate a sound and complete exact model of its solution space.

2.7 Runtime Analysis

This analysis is straightforward and does not take into consideration the many ways that the algorithm could be made more efficient in practice such as both parallel computation (possible with classical and actual quantum systems) and data partition strategies that reduce the number of structures that would be compared with each other.

Since X structures cannot have more (width) than 5 elements, we can have no more than n^5 distinct X structures overall. If we compare each X structure to every other X structure during first-generation we get n^5 by n^5 total comparisons giving us a total of n^{10} comparisons. All modifications are constant time and so our runtime cannot exceed that needed to make individual comparisons, giving us a maximum of n^{10} total number of operations.

For second generation, if we use only the recommended basis set, similarly, we get no more than n^3 possible X structures. Here, we do not compare structures but instead loop over their collection for a possible maximum of n times to derive a solution. This gives us a possibility for no more than n^4 possible operations.

Considering the totality of all possible operations, we can see that our algorithm has a maximal runtime complexity of $O(n^{10})$.

3. Conclusion

We have introduced a quantum algorithm that runs on classical computers, a first from what we know. We hope that this would prove useful to mathematicians and physicists who aim to study quantum systems and their related mathematical structures and properties. We feel that the greatest contribution we would be able to make in addition to being able to solve the class of problems currently known as NP in the computing literature efficiently (and we hope to have successfully proved is P is equal to NP) is in the scalable and iterative modelling of quantum systems.

We conjecture that problems with existing rational order – mathematical and physical, would be particularly tractable using our algorithm as they would already have some intrinsic quantum computational/algorithmic patterns embedded in their structure. In particular, this should help us better understand quantum systems and their efficient design and improvement. We hope that this spurs a new era in work on both quantum and classical algorithms and systems as we better understand their properties and possibilities on all possible computing platforms.

References

[1] Valiant, Leslie (17–19 October 2004). Holographic Algorithms (Extended Abstract). FOCS 2004. Rome, Italy: IEEE Computer Society. pp. 306–315. doi:10.1109/FOCS.2004.34. ISBN 0-7695-2228-9.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In Proceedings of the 3rd ACM Symposium Theory of Computing, pages 151–158, Shaker Heights, Ohio, 1971.

[3] Leonid A. Levin. Universal search problems. Problemy Peredachi Informatsii, 9(3):265–266, 1973.

[4] Richard M. Karp. Reducibility among combinatorial problems. Complexity of Computer Computations, (R. Miller, J. Thatcher eds.), pages 85–103, 1972.